A practical guide to C++

Janis Fehr
fehr@informatik.uni-freiburg.de

SommerCampus2004

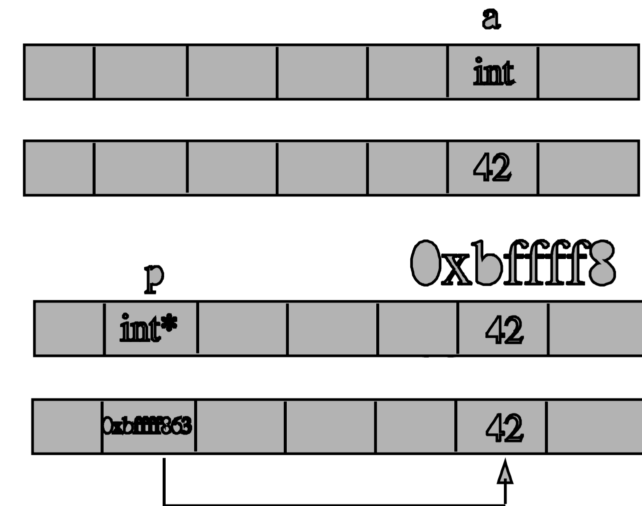
Kurstag 2: Agenda

- Wiederholung der Themen von Tag 1
- Codeoptimierung
  - Namespace
  - Typedef
- Type Cast
- **Debugging**
  - gdb
  - Valgrind
- **Objektorientiert Programmieren in C++ (1)**
  - einfache Klassen
- Codeorganisation
  - Header Files
  - Makefile
- **Objektorientiert Programmieren in C++ (2)**
  - Vererbung
  - komplexere Klassen

Wiederholung:

Zeiger:

```
int a;

a = 42;

std::cout<<&a;

int* p;

p = &a;
```



Streams:

```
std::ifstream file(''data.txt'');//öffne data.txt zum Lesen
std::string word;

file >> word;
```

Namespace:

Namespaces dienen hauptsächlich der Übersichtlichkeit. Sie zeigen die Zusammengehörigkeit von Klassen, Strukturen, Variablen usw.

Wir haben schon einen Namespace kennengelernt, den der C++ Standard Bibliothek: **std** z.B. Bei `std::cout`

Meist befindet man sich nur in wenigen Namespaces, daher kann man diese auch vorab einbinden:

```
using namespace std;

int main()
{
   cout<<''Hello World''<<endl;
...
```

Man kann sich auch leicht einen eigenen Namespace definieren:

```
namespace MySpace
{
   int foo()
...

MySpace::foo()
```

Typedef

Mit typedef lassen sich alias Namen zu Typen frei definieren. Dies dient hauptsächlich der Übersichtlichtkeit und Lesbarkeit (man sollte es nicht übertreiben !) des Codes.

```
Typedef *int Zeiger_auf_Int;

...

int a =32;
Zeiger_auf_Int b;
b = &a;
```

Verwendung aber meist nur bei komplexen Typen:

```
typedef svt::FVwithMultiClassCoefs<svt::SparseFV> FV;
```

Type Cast

Impliziter Cast:

Kompatible Typen werden bei Zuweisungen automatisch gecastet:

```
int a; double b;
a = b; //automatischer cast
```

Expliziter Cast:

Typen können auch manuell gecasted werden:

```
int a; double b;
b = (double) a; //manueller cast
```

Cast bei Zeigern:

Zeiger haben einen festen Typ auf den sie zeigen. Dieser kann aber auch verändert werden:

```
int* a; double b;
reinterpret_cast<double*> a = &b; //Zeiger cast
```

Debugging

"it's not a bug - it's a feature !" (Bill Gates)

Debugging ist in jeder Programmiersprache wichtig - in C++ aber ganz besonders...

Viel Macht (über der Speicher) bedeutet, dass auch viel Falsch (ge)Macht werden kann: **Speicherverletzungen.**

```
int* a = new int[5];
int b;
...
b=a[7];
```

Segmentation Fault !

gdb

- GPL Debugger - quasi Standard
- Features:
  - Breakpoints
  - schrittweise Ausführung
  - Beobachtung von Variablen
  - Zeigt Speicherinhalt
  - Analyse bei Abstürzen
- Benötigt Informationen vom Compiler
  - kompeliere mit der Option `-g`

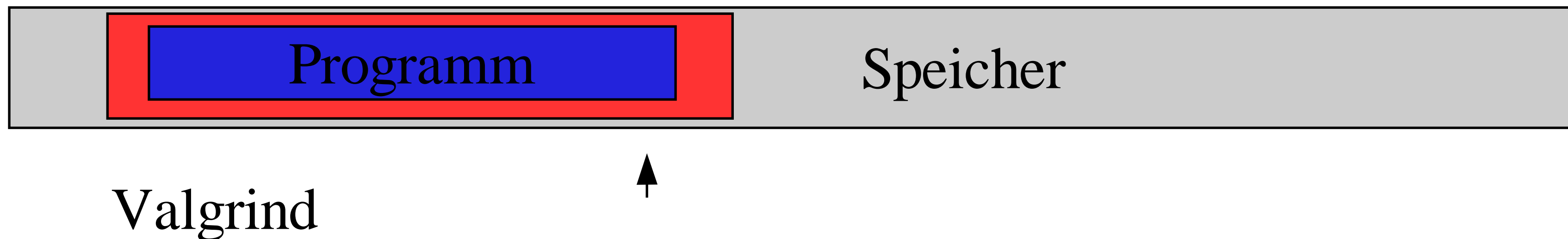
```
g++ -g debug.cc -o debug
gdb debug
(gdb) run
```

C++

Valgrind

- Sehr schönes Tool zum Fiden von Memory-Leakes
    - Findet auch Fehler die keinen, SegV verursachen. (gdb nicht)
    - Läuft leider nicht auf den Suns -> in Asembler geschrieben

Funktionsweise:



Demo

Objekt orientiert programmieren

```
Class MyClass
{
      public:
            MyClass();
            ~MyClass(){};
            void setA(int input);
            int getA();
      private:
            int a;
};


MyClass::MyClass(){a=5;}

void MyClass::setA(int input)
{
   a=input;
}

int MyClass::getA()
{
   return a;
}
```

Klassenname

Konstruktor

Default Destruktor

Öffentliche Methode ohne Rückgabewert

Private Variable

Implemnetation des Konstruktors

Implementation der Methoden

Instanzieren von Klassen

```
int main()
{
        MyClass Instanz1;

        cout<<Instanz1.getA()<<endl;
        Instanz1.setA(42);
        cout<<Instanz1.getA()<<endl;
        ...
```

Erzeuge Instanz von MyClass

Rufe Funktionen der Instanz auf

Die Instanz liegt auf dem Stack !

```
        MyClass *Instanz2 = new MyClass();

        cout<<Instanz2->getA()<<endl;
        Instanz2->setA(42);
        ...
```

Besser !

Eigenschaften von Methoden

Nebem **public** oder **private** gibt es für Methoden noch Weitere Eigenschaften, **const** haben wir schon kennen gelernt... bleibt vorerst noch **static:**

```
Class MyClass2
{
   public:
      static int counter;
      MyClass2(){counter++};
      ~MyClass2(){counter--};
   ...
}
```

Static Methoden und Variablen "gehören" der Klasse und nicht der Instanz.

d.h. Es existiert nur eine Kopie die sich alle Instanzen Teilen !

```
cout<<MyClass::counter<<endl;
```

Code Organisation

Bei Projekten wird die Definition der Klassen von der Implementation getrennt.

MyClass.hh

```
#include<iostream>

Class MyClass
{
   public:
      int foo(int a);
      ...
   private:
      ...
};
```

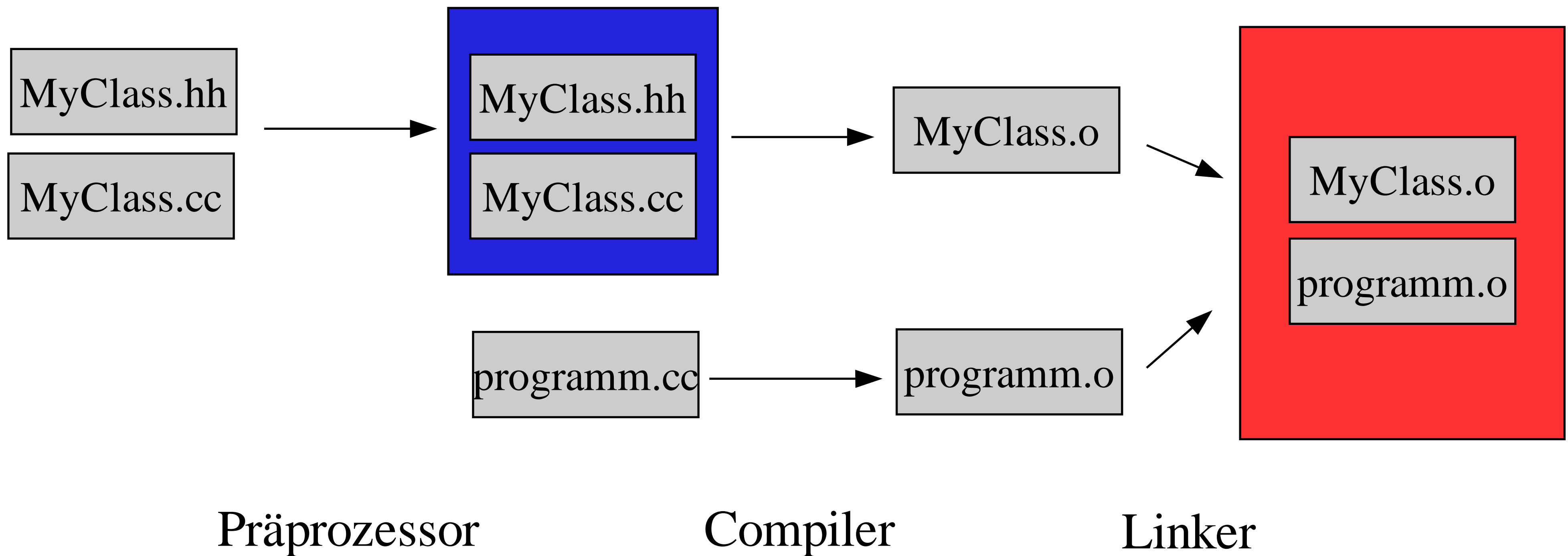
MyClass.cc

```
#include ''MyClass.hh''

int MyClass::foo(int a)
{
         ...


}
```

programm.cc

```
#include ''MyClass.hh''
int main()
{
   MyClass a = new MyClass();
```

Wie fügen sich die Teile zusammen ?



Weil man das alles nicht von Hand machen will, benutzt man ein **Makefile**

Das Makefile:

Makefile:

```
CXX = g++
CXXFLAGS = -Wall -g

all: programm
```

Compiler festlegen

Compileroptionen

Gib Infos an gdb

Zeige alle Fehler/Warnungen

Was soll kompiliert werden ?

Kompelieren und Linken durch eingabe von : **make**

Beeinflussung des Compilers

1. Funktionen **inline** definieren:

```
Inline int min(int a, int b);
...

cout<<min(6,8)<<endl;
```

Schnell, aber nur für kleine Funktionen zu empfehlen.

2. mit **Macros** den Compilierprozess steuern:

```
#ifndef MACRO_NAME
   #define MACRO_NAME

   #include ''MyClass.hh''

   class XY
   ...
#endif
```

z.B. Darf ein Headerfile nur ein mal eingebunden werden

C++

Objekt orientiert programmieren (2)

Vererbung:

```
class Parent
{
   public:
      void getA();
      int foo();
      ...
   private:
      ...
}

class Child : public Parent
{
   public:
      void getB();
      int foo();
   private:
      ...
}
```

Basis Klasse

Vererbung

Zugrifsrechte auf die geerbten Methoden

Eigene Methode

Überschreibe geerbte Methode

Zugriffsrechte auf geerbte Methoden

```
Class Parent
{
   public:
      ...
   protected:
      ...
   private:
      ...
}

class child : Zugriffsrecht Parent
{
   ...

}
```

Recht	Parent	Child
public	public	public
	protected	protected
	private	-
protected	public	protected
	protected	protected
	private	-
private	public	private
	protected	private
	private	-

Wird das Zugriffsrecht nicht gesetzt, nimmt der Compiler **private** an.

Mehrfachvererbung

Eine Klasse kann von meheren Basisklassen erben:

```
class Child : public Parent1, private Parent2
{
    ...
```

Überladen des Konstruktors

Die Argumente des Konstruktors der Basisklasse müssen auch nach der Vererbung übergeben werden.

```
class Parent
{
    public:
        Parent(int a, int b);
...

class Child : public Parent
{
    public:
        Child(int a, int b, float c);
```

Project

- Schreibt eine Klasse Pixel mit folgenden Eigenschaften:
  - Private RGB Werte
  - Methoden um die Werte zu setzen und abzufragen
  - Konstruktor: default Wert ist schwarz
- Schreibt eine Klasse BinaryPixel die von Pixel erbt und erweitert sie:
  - Die die **L**east**S**ignificant**B**its der RGB Kanäle können ausgelesen
  - und verändert werden.

Implementiert eine Hauptrutine welche die Funktionen testet und organisiert den Code in Header, Objekt und Programm Files.